NAME: _____ *Work on your own. Write in your answers (neatly) for each section. If we are busy with another student, flag us and then move on to the next section until we can get to you.*

(-1) Declare variables for all the built-in types: `char, short, int, long, float, double`. Use the **`sizeof()`** macro to print the amount of memory required for one(1) variable of each type. Of course, you must give the units for your values. Show me your code and write your answers below.

`char` _____ `short` _____

`float` _____ `int` _____

`double` _____ `long` _____

(0) In `main()`, declare and initialize an array of (`short`) values. The quickest way to do this is to use auto-initialization while declaring your array:

```
short A[10] = {34, -7, 4, 3, 88, 19, 21, -7, 9, 0};
```

(1) **Draw a picture of `A` in memory** below. Note: since you declared this array *in* your `main()` function, the array `A` will be located on the runtime stack (not in the dynamically allocated heap, right?). Label your array boxes with both the index value *and* the memory address (see below).

(2) Determine the actual memory (address) location of the **initial cell** in the array. Remember that you can print the address of any variable or array cell:

```
cout << "Memory location of A:   " << &A[0] << endl;
# Note:  &A[0] can also be written as just the name of the array, A

cout << "Value in array here is: " <<  A[0] << endl;
```

(3) Based on the value that you determined in (2) above, **label all the other memory locations** in your picture in (1) above. (You can just use the last three hexadecimal digits).

(4) Implement a for-loop to print the **memory address of all the cells** in the array. This should confirm your answers from (3) above, right?

(5) Write another loop to do the same work as (4) above, but this time, do *not* use array indexing (that is, do *not* use `&A[i]`). Rather, use one pointer (obviously, a pointer to a `short`). Use pointer arithmetic to move this one point down the array and print the memory locations and values at each cell in the array `A`.

```
short *pA;
for(…
```

Call us over to see your results so far when you reach this point.

(6) Write one loop to **reverse the values (in place) in the array**. Your solution should use two pointers

```
short* front;
short* back;
```

Print the values in the array *before* and *after* your loop.

Call us over to see your results so far when you reach this point.

(7) **Practice Exam Questions**

Consider the following collection of declarations:

```
short i      = 43;
short *ip    = &i;
short V[3] = {3, 14, 5};
short *vp  = V;
short *ap  = 0;
```

(A) Show a picture of memory, being careful to label each variable with an **address**, **name** and **contents**. (just make up memory locations).

(B) Using your memory locations, give the value of each of the following expressions.

`*ip`	_____	`*(vp+1)`	_____
`&V[4]`	_____	`*vp + 1`	_____
`vp[2]`	_____	`vp + 3`	_____
`*vp`	_____	`*ap`	_____

(8) Implement the code from (7) above. Using *nicely labeled output*, print the values in (7B) above. Check your answers. Any surprises?

Call us over to see your results so far when you reach this point.

(9) Imagine you have the following C++ function declaration `swap()`. This function swaps the values in the two pass-by-reference arguments. Note the `&` indicates the arguments are pass by reference. Note the "Pre-Post" documentation that tells the reader that the arguments take "**in**" values and also send new values back "**out**" to the calling routine.

```
void swap( /* in-out */ short& one,   /* in-out */ short& two);
```

Implement the swap function and test it by calling it from `main()`. Obviously, in `main()` you should print out the two values both before and after calling `swap()`.

(10) Imagine you are working on an embedded device that does *not* have a C++ compiler *(say "**dang**" out loud so I know you have reached this point of the lab)*. However, the device does have a C compiler. Rewrite the swap function to work in C (but you can still use your C++ project/compiler, right?). *Note: the C language does not allow pass by reference, but of course, it does use pointers.* Rewrite the C++ function above as an alternate called `Cswap()` where the two arguments are pointers to `short`s. Test your *new function* `Cswap()` in `main()` as you did in (9) for the C++ version. The new function declaration (sometimes called a function "prototype") is shown below:

```
void Cswap( short* one,  short* two);
```

Call us over to see your results so far when you reach this point.

---

(11) **Shallow *vs.* Deep copy**: making a "safe" Copy CTOR.

Read this material; it is quite well written (imho)
http://www.learncpp.com/cpp-tutorial/912-shallow-vs-deep-copying/

(a) Did you read that web page above? Please read it now.

(b) Download the Starter Kit from onCourse for this lab.

(c) **Implement the behaviors listed in the header (.h) for this class Tweet**. At this point, *only* implement the methods listed in the public part of the class declaration (that is, do *not* implement the copy CTOR yet).

Tweet.h

```
class Tweet{
 public:
    Tweet();
    Tweet(char* msg);

    ~Tweet();   // DTOR

    // get tweet from stdin
    friend istream& operator>>(istream& fin,  Tweet& T);
    // dump tweet to stdout
    friend ostream& operator<<(ostream& fout, const Tweet& T);

private:
    static const short MAX_TWEET;

    char* theTweet;
};
```

Call us over when you have the main() program running.
Can you describe what is happening?

(d) Write terse, precise *(neat)* **definitions** in English for the following:

A shallow copy is

A deep copy is

(e) Implement a **copy CTOR** to fix the shallow copy problem.

(f) What other methods should you write to ensure this class works "safely"?